

Sorting Algorithms

Written by:

Artiom Leliuhin, 1st year of IE

Email - artiom.leliuhin00@e-uvv.ro

GitHub - <https://github.com/ALeliuhin>

Plan:

1. Introduction – (page 2)
2. Main Concepts – (page 3 – 13)
3. Testing – (page 13 – 19)
4. Conclusion – (page 20)
5. Bibliography – (page 21)

Introduction

In the 1st semester of this academic year, I got familiar with sorting algorithms at the Algorithms and Data Structures course. We were discussing a lot of things related to the complexity of algorithms, their growth rate and how these algorithms are applied nowadays, distinguishing by their advantages and ability to sort complex data.

The most interesting thing about sorting methods for me, however, is the theory behind each of those algorithms and how I could possibly implement those on my own. The first semester in the discipline Computer Science imposed me with the special ability to reason about problems that I solve, keeping in mind the resources required to solve problems, the speed and stability of the solutions.

That is why in this paper I would like to compare the methods that we studied so far by both theory and implementation perspectives. We will be looking at the following sorting algorithms:

- Quicksort
- Merge Sort
- Insertion Sort
- Selection Sort
- Bubble Sort
- Timsort (Python built-in `sorted()` method)

Main Concepts

Quicksort

Quicksort is one of the most widely used algorithms for sorting arrays because of its speed and ability to be modified.

The Idea:

- I. A random element referred to as a pivot element is picked (usually the first or the last).
- II. Distributing the elements happens in a way that the elements that are less than a pivot element are placed on the left side of the pivot, and those that are greater on the right, respectively.
- III. Recursively **step I** and **step II** are applied for the left and the right part of the array until the whole array is completely sorted.

Python Implementation:

```
5  def quicksort(array):
6      if len(array) <= 1:
7          return array
8      else:
9          pivot = array[0]
10
11         left = [x for x in array[1:] if x < pivot]
12         right = [x for x in array[1:] if x >= pivot]
13
14         return quicksort(left) + [pivot] + quicksort(right)
15
```

Complexity:

- Best Case: $O(n \log n)$.

The best-case scenario occurs when choosing “a good pivot” which means that the input array is already partially or fully sorted.

- Average Case: $O(n \log n)$.

The average-case scenario is the most likely scenario for unsorted arrays.

- Worst Case: $O(n^2)$.

The worst case occurs when choosing the greatest or the smallest object in the array, that subsequently leads to unbalance distributing of elements.

Stability:

Quicksort is not a stable method since the relative order of equal elements in the array may change after sorting.

Conclusion:

Overall, the best case and the worst case happen very rarely in practice, that gives the algorithm an acceptable complexity of $O(n \log n)$. Despite many disadvantages (ex. instability), as mentioned above, Quicksort served as a great standing point for modifications such as Timsort, that we shall discuss later.

Merge Sort

Merge Sort is one of the most efficient and popular algorithms nowadays, that is considered a good example of “Divide and Conquer” principle.

The Idea:

- I. The algorithm divides an input array into two halves.
- II. Recursively the halves are divided until reaching a two-element array or one-element array.

- III. The elements in those small arrays are compared between each other and combined until reaching the final state, when the array is sorted.

Python Implementation:

```
17     def merge_sort(array):
18         if len(array) <= 1:
19             return array
20         else:
21             mid = len(array) // 2
22             left = array[:mid]
23             right = array[mid:]
24
25             merge_sort(left)
26             merge_sort(right)
27
28             i = 0
29             j = 0
30             k = 0
31
32             while i < len(left) and j < len(right):
33                 if left[i] <= right[j]:
34                     array[k] = left[i]
35                     i += 1
36                 else:
37                     array[k] = right[j]
38                     j += 1
39                 k += 1
40             while i < len(left):
41                 array[k] = left[i]
42                 i += 1
43                 k += 1
44             while j < len(right):
45                 array[k] = right[j]
46                 j += 1
47                 k += 1
48             return array
```

Complexity:

- Best Case: $O(n \log n)$.

The best-case scenario occurs when the array is already sorted or when the elements are equal.

- Average Case: $O(n \log n)$.

This is the most frequently encountered case when the objects inside the array are randomly ordered.

- Worst Case: $O(n \log n)$.

The worst case occurs when the array is sorted in descending order.

Stability:

Merge Sort is a stable algorithm since the order of equal elements is persisted.

Conclusion:

In general, Merge Sort is a great option for handling big data, making it a popular choice. It is stable, fast and efficient because of its complexity of $O(n \log n)$ that remains the same in all cases. The interesting thing is that the algorithm was invented by John von Neumann himself in 1945.

Insertion Sort

Insertion sort is one of the fastest and simplest algorithms for sorting small arrays. It works very similar to sorting cards in a bridge hand.

The Idea:

- I. Each element beginning with the second (auxiliary element) is inserted into the preceding array of elements.

- II. The inserted element finds its place depending on its value.
- III. The process repeats until the algorithm iterates through the entire array, placing its items before the auxiliary element.

Python Implementation:

```
65 def insertion_sort(array):
66     if len(array) <= 1:
67         return array
68     else:
69         for i in range(1, len(array)):
70             key = array[i]
71             j = i - 1
72             while j >= 0 and key < array[j]:
73                 array[j + 1] = array[j]
74                 j -= 1
75             array[j + 1] = key
76         return array
```

Complexity:

- Best Case: $O(n)$.

The case occurs when the input array is already fully sorted, making it easy to iterate through and keeping each auxiliary element at the same position.

- Average Case: $O(n^2)$.

The most common case when the elements are randomly ordered. However, $O(kn)$ case may occur for partially sorted arrays where k – the number of elements from the sorted position.

- Worst Case: $O(n^2)$.

The worst-case scenario happens when the array is sorted in a descending order.

Stability:

Insertion Sort is a stable algorithm.

Conclusion:

Insertion Sort is an easy to implement stable algorithm, but the average and the worst complexity of $O(n^2)$ makes it slow for big data comprehension. However, it is still very fast for small data and the furthermore tests will show that.

Selection Sort

Selection Sort is a very simple algorithm for sorting small arrays.

The Idea:

- I. The smallest element in the array is found and interchanged with the i-element of the array.
- II. The process is repeated until reaching the last element of the array, that will be the greatest one.

Complexity:

- Best Case: $O(n^2)$.

The best case occurs when no swaps are performed, that implies that the array is already sorted.

- Average Case: $O(n^2)$.

The average case happens when the array is randomly sorted, invoking an n-th number of swaps.

- Worst Case: $O(n^2)$.

In the case of selection sort, the average and the worst case are very similar since the number of swaps and iterations may slightly vary.

Stability:

Unstable, doesn't persist an order of relatively equal elements.

Python Implementation:

```
51  ✓ def selection_sort(array):  
52      if len(array) <= 1:  
53          return array  
54  ✓  else:  
55  ✓      for i in range(len(array)):  
56          min_index = i  
57  
58  ✓      for j in range(i + 1, len(array)):  
59          if array[j] < array[min_index]:  
60              min_index = j  
61          (array[i], array[min_index]) = (array[min_index], array[i])  
62      return array
```

Conclusion:

Overall, Selection Sort may be a good option for educational purposes because of its easy implementation. However, its complexity of $O(n^2)$ makes it really slow and unpractical when it comes to sorting a bunch of objects.

Bubble Sort

Bubble Sort is a very popular and simple algorithm that works by interchanging the pair of elements in order.

The Idea:

- I. The algorithm iterates through the adjacent elements, interchanging them when $j > j + 1$
- II. The process is repeated $n - 1$ times until no elements are interchanged.

Python Implementation:

```
79 def bubble_sort(array):
80     if len(array) <= 1:
81         return array
82     else:
83         n = len(array)
84         for i in range(n):
85             for j in range(0, n - i - 1):
86                 if array[j] > array[j + 1]:
87                     array[j], array[j + 1] = array[j + 1], array[j]
88         return array
```

Complexity:

- Best Case: $O(n)$.

The best-case scenario for bubble sort occurs when the input array is already sorted. In this case, bubble sort passes through the array only one time and performs no swaps.

- Average Case: $O(n^2)$.

The average case is common when the array is unordered, leading to a certain number of swaps.

- Worst Case: $O(n^2)$.

The worst-case scenario happens when the array is sorted in descending order, making the algorithm perform the maximum number of swaps.

Stability:

Bubble Sort is a stable algorithm, provided that the strict inequality is used: $x[j] > x[j + 1]$

Conclusion:

Bubble Sort is good for educational purposes, that's how it became popular among newcomers. However, it's completely impractical and nowadays data analytics uses only the derivatives of this algorithm (ex. Heapsort).

Timsort

Timsort is a Python built-in sorting method in Python and Java that was derived from quicksort and merge sort and is considered very fast and efficient for various types of objects. It would be very interesting to add this to our tests.

The Idea:

- I. Exploits the existing order in the data to minimize the number of comparisons and swaps that is achieved by dividing the array into small parts called runs.
- II. Defines the size of one run.
- III. Merges the runs and doubles the size of the run until it exceeds the array's length.
- IV. Merges until the array is completely sorted.

Python Implementation:

```
91 def built_in_timsort(array):  
92     return sorted(array)
```

For Python implementation it is enough to return `sorted(array)` since the method `sorted()` is based on Timsort.

Complexity:

- Best Case: $O(n)$.

In this case, Timsort can take advantage of its merge sort component to efficiently merge these subsequences while minimizing unnecessary comparisons and swaps. Timsort can detect runs (subsequences of the array that are already sorted) and merge them without further comparisons or swaps.

- Average Case: $O(n \log n)$.

The average-case scenario can be achieved when applying merge sort on large runs and insertion sort on smaller runs.

- Worst Case: $O(n \log n)$.

The worst-case scenario occurs when the input arrays is sorted in a descending order. However, it can handle worst-case scenarios like reverse-sorted input by breaking the array into the runs and combining them.

Stability:

Timsort offers stability and that is why it was picked for comprehending different types of objects.

Conclusion:

In conclusion, Timsort is a highly efficient and versatile sorting algorithm that performs well in a wide range of scenarios. Its adaptiveness and hybrid design make it well-suited for sorting real-world data efficiently.

Overview

Method	Best case	Average case	Worst case	Stability	Weight
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Unstable	7
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Stable	11
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Stable	8
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable	4
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Stable	8
Timsort	$O(n)$	$O(n \log n)$	$O(n \log n)$	Stable	12

RED – 1, **YELLOW** – 2, **GREEN** – 3

By making the weight system and calculating the weights, the winner is Timsort (12), then goes Merge Sort (11), both Insertion Sort (8) and Bubble Sort (8), Quicksort (7) and the slowest horse – Selection Sort (4).

Testing

In this part of the work, I would like to finally test the algorithms by passing the same randomly generated array of integer numbers and calculating execution time for each sorting function.

First, let me introduce some important notes that affect the desired outcome and may considerably vary:

1. Operating system:

Since different operating systems use different architecture, software, and packages, this may lead to unexpected results when sorting on different systems. For this experiment specifically I will be using Windows 11.

2. Hardware:

Hardware plays a crucial role in the execution time for running programs. The most important ones are CPU and RAM. For this experiment I am using a laptop with Intel Core i7-13700H up to 5GHz of frequency and 16gb RAM.

3. Programming language:

The programming language used is also very crucial, since choosing the one may depend on the task and desired result. The low-level ones are significantly faster, but in our case, I will be using Python v.3.12.0 since we studied it in the 1st semester.

4. Implementation:

Implementation of the program and, as here, sorting functions may also lead to different results due to various built-in programming methods, list comprehensions and the number of variables to store the results.

5. Objects to sort:

Objects that are sorted is a very important note to consider. Obviously, sorting n-th number of strings and n-th number of integers would give us a significant difference, since strings are basically arrays and stored in our memory as arrays. But for this experiment we are using only integers.

From now on, I will quickly describe how my source program works (the link to the GitHub repository will be provided):

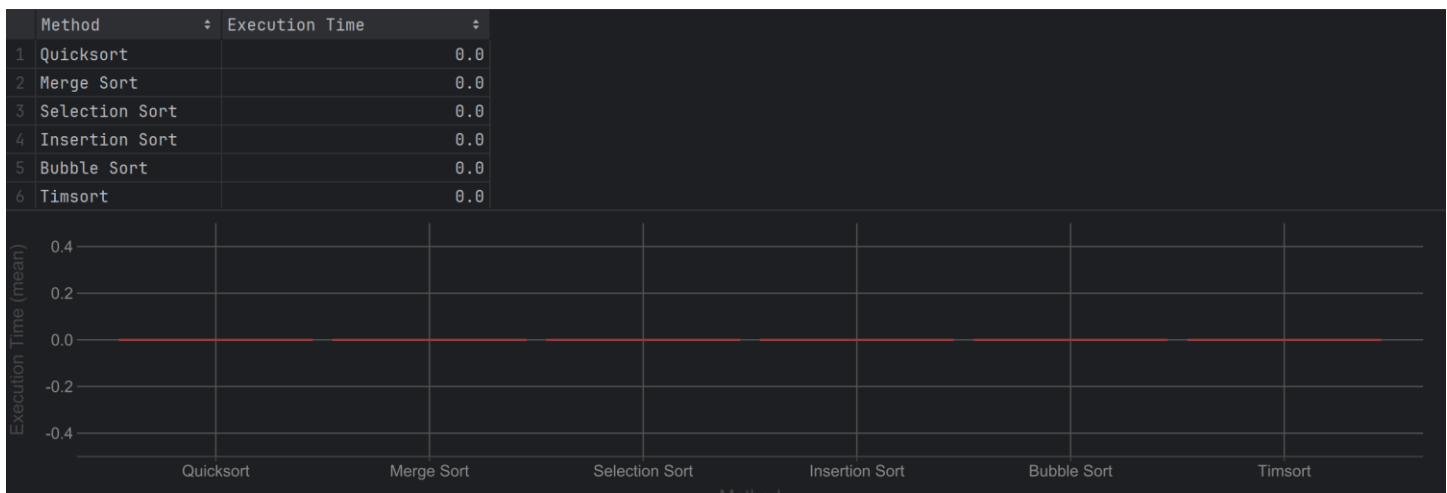
- I. The user inputs the number of integers for the array.
- II. The user inputs the limit as a value for each integer (it will be considered as $[-n, n]$)

- III. Depending on the input values, the array will be generated using the “random” module. We would eventually have an array of k-integers, each being within the $[-n, n]$ limit.
- IV. For each sorting function, the generated array will be passed, and the execution time will be calculated using the “time” module. All the results are going to be stored in a dictionary.
- V. The dictionary is sorted by execution time and the output will be written into a CSV file, containing the fields “Method” and “Execution time”.

We will keep the weight system to draw a conclusion in the end. Now we can start testing.

(Execution time will be calculated **in seconds**)

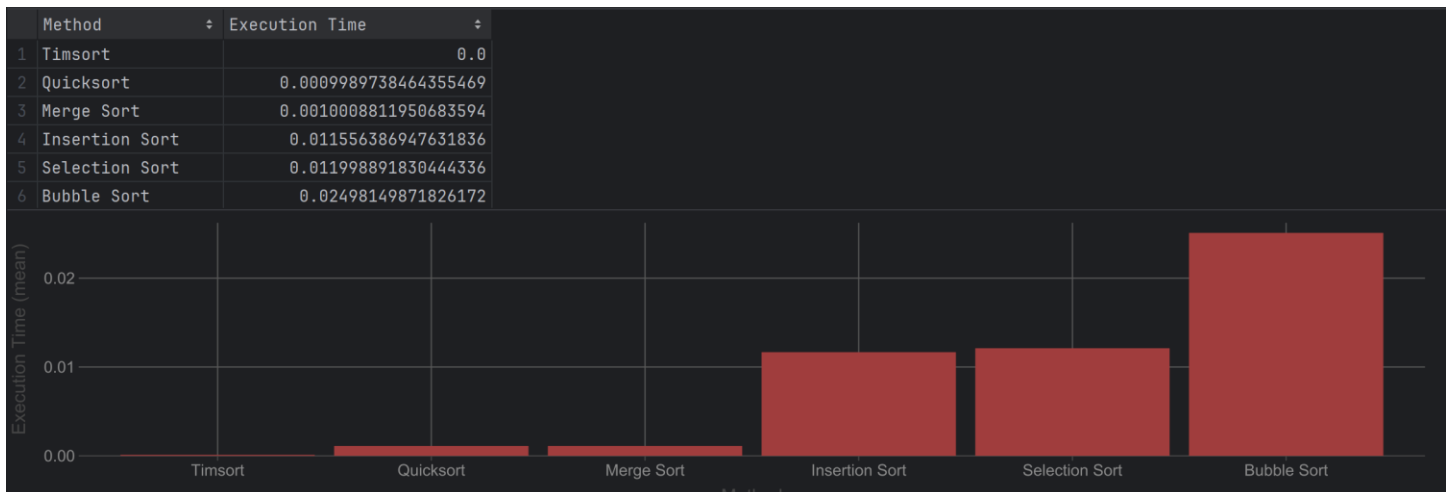
Test-1: 100 elements, $[-10000, 10000]$



The first test showed that all the sorting functions did their job quite fast and for 100 elements the amount of time required was unnoticeable. So, everyone is getting green for their efforts.

Quicksort	Merge Sort	Insertion Sort	Selection Sort	Bubble Sort	Timsort
-----------	------------	----------------	----------------	-------------	---------

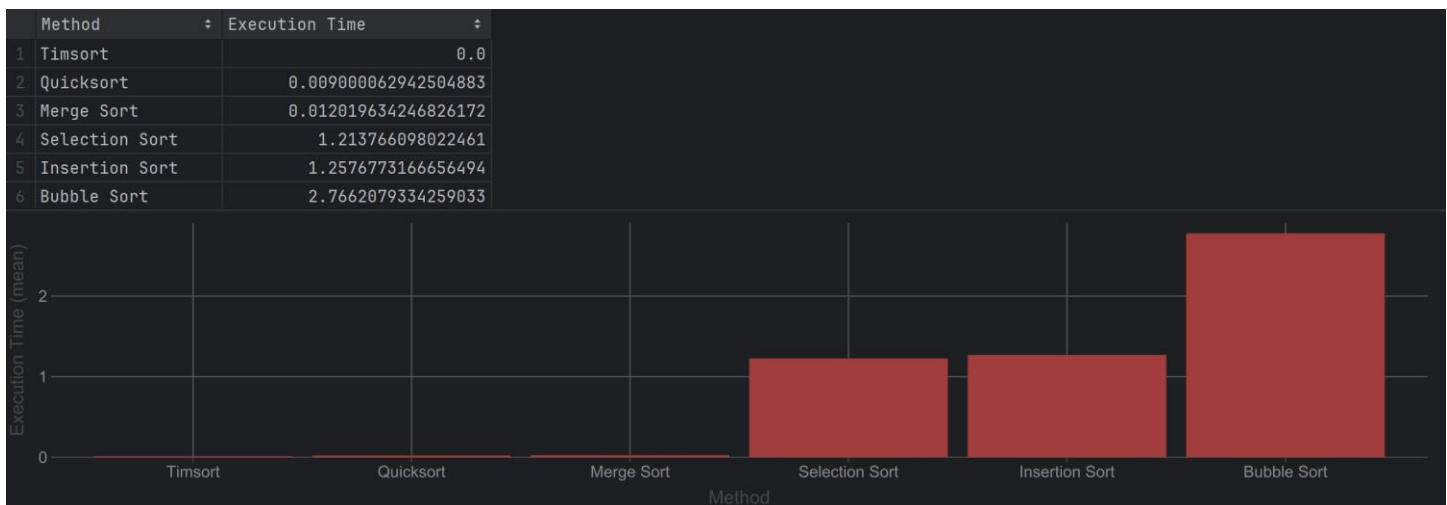
Test-2: 1000 elements, [-100000, 100000]



Interesting, but here we already see that both Insertion Sort and Selection Sort took 10 times bigger amount of time than Merge Sort. Shall we give them a chance and give yellow instead of red? For now, we will forgive them.



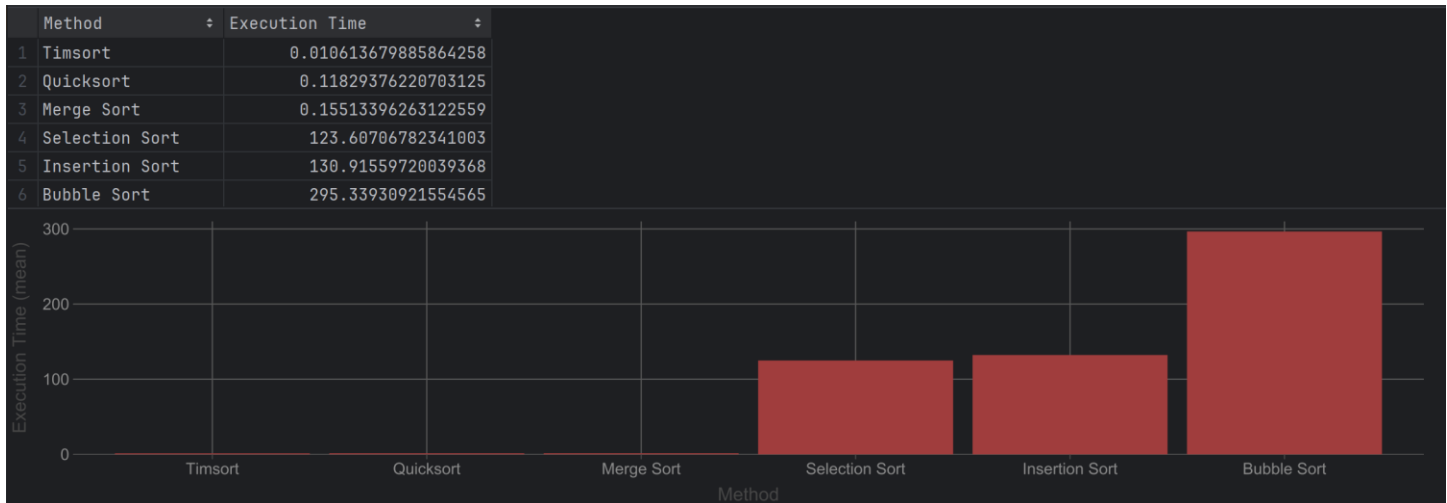
Test-3: 10000 elements, [-1000000, 1000000]



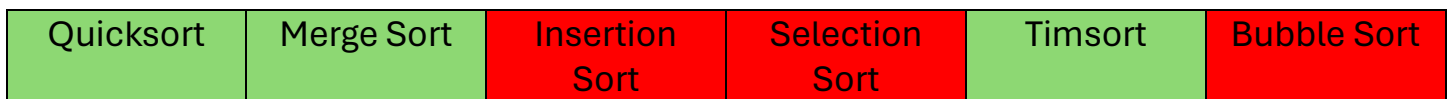
As the problem size grows, the $O(n^2)$ guys start feeling worse, don't they?



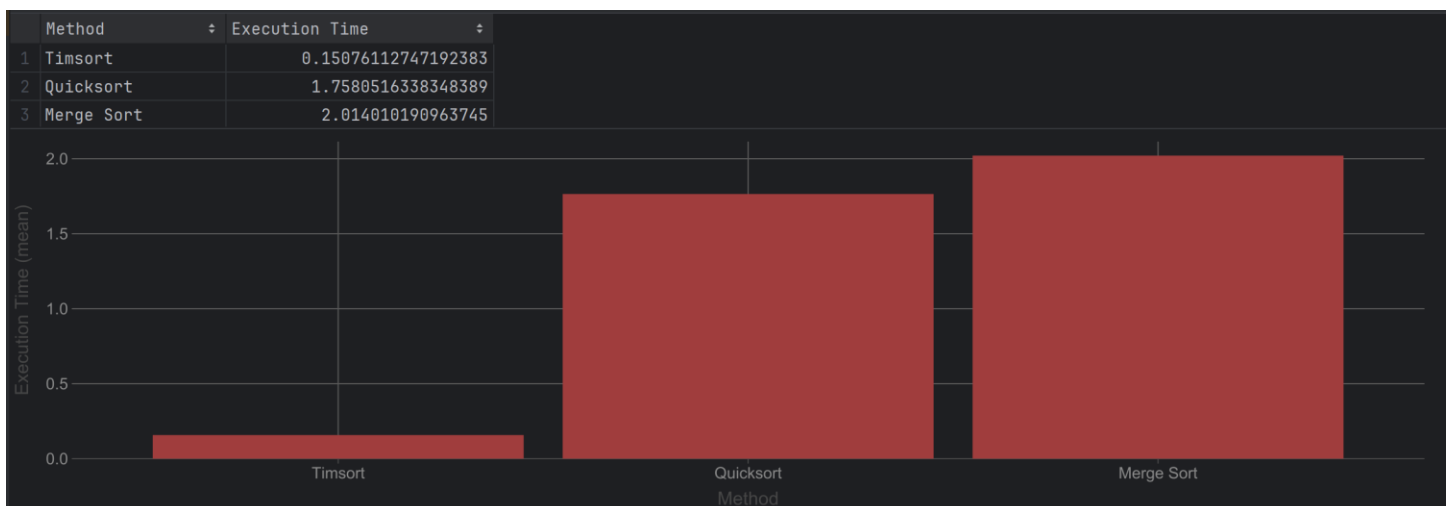
Test-4: 100000 elements, [-10000000, 10000000]



Now things have become serious. The fans are spinning, RAM usage arises, and the obvious losers take 2-5 minutes to sort. From now on and for the sake of my machine, Selection Sort, Insertion Sort and Bubble Sort will be taken aside and set red for each subsequent test.



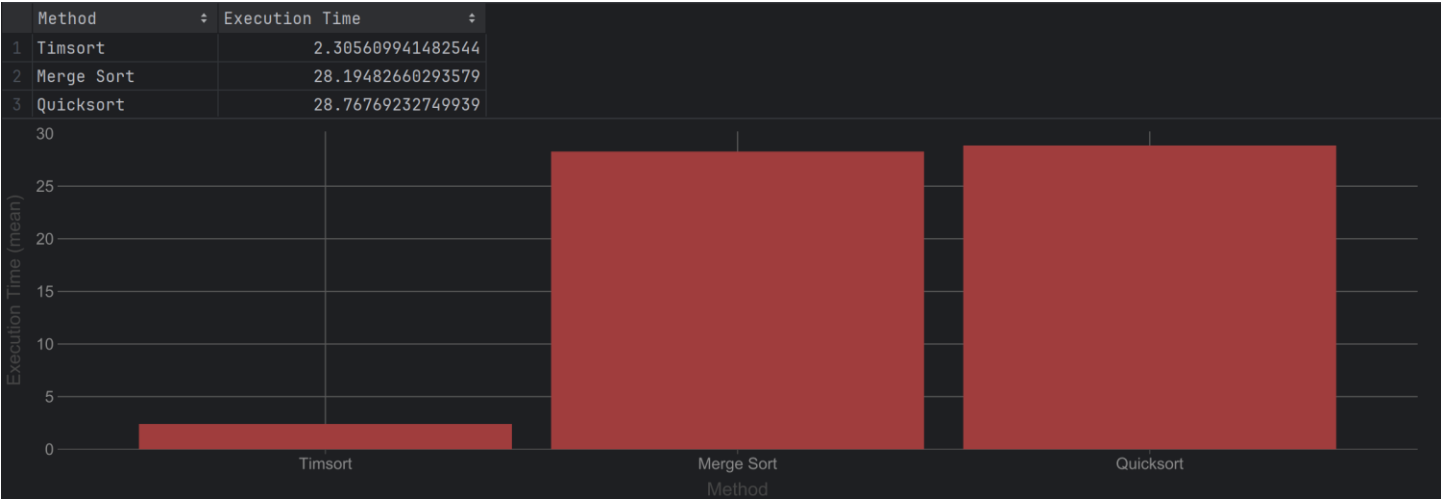
Test-5: 1000000 elements, [-100000000, 100000000]



Fast. But not enough. Both Merge Sort and Quick Sort will be given yellow.

Quicksort	Merge Sort	Insertion Sort	Selection Sort	Timsort	Bubble Sort
-----------	------------	----------------	----------------	---------	-------------

Test-6: 10000000 elements, [-1000000000, 1000000000]

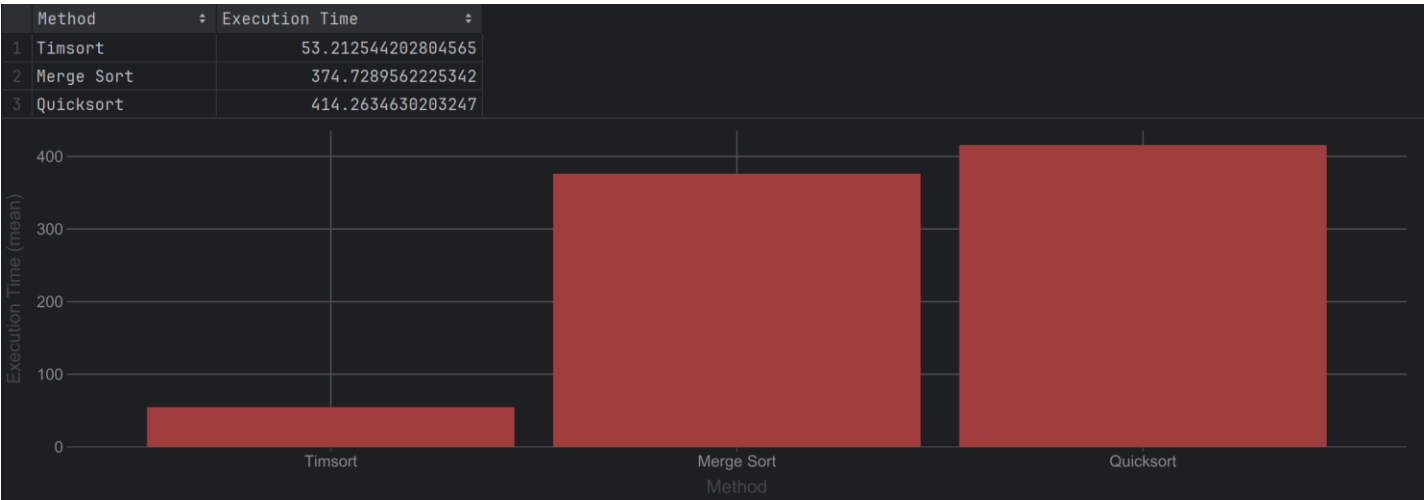


Quicksort started losing its positions against Merge Sort, but Timsort is still doing great!

Quicksort	Merge Sort	Insertion Sort	Selection Sort	Timsort	Bubble Sort
-----------	------------	----------------	----------------	---------	-------------

Test-7: 100000000 elements, [-10000000000, 10000000000]

Name	Status	21% CPU	93% Memory
> PyCharm (6)		12.3%	9,140.8 MB




It was a very long time to wait, but from here we will get rid of Merge Sort and Quicksort, and we will be comparing only Timsort.

Quicksort	Merge Sort	Insertion Sort	Selection Sort	Timsort	Bubble Sort
-----------	------------	----------------	----------------	---------	-------------

Test-8.1: 1000000000 elements, [-1000000000000, 1000000000000]

Unfortunately, in Test-8 I got a Memory Error, and it didn't allow my program to be executed. I shall try Test-8 with twice as few elements.

Test-8.2: 500000000 elements, [-1000000000000, 1000000000000]

Name	Status	9% CPU	95% Memory	5% Disk	0% Network
>  PyCharm (6)		7.9%	11,183.0 ...	9.6 MB/s	0 Mbps

This test also failed, even though I was waiting for 4 hours. The array is too big. We will not consider the Test-8 for the results.

Overview

Method	Test-1	Test-2	Test-3	Test-4	Test-5	Test-6	Test-7	Overall
Quicksort	~0.0	~0.00099	~0.00900	~0.11829	~1.75805	~28.7676	~414.263	16
Merge Sort	~0.0	~0.00100	~0.01201	~0.15513	~2.01401	~28.1948	~374.728	16
Insertion Sort	~0.0	~0.01155	~1.25767	~130.915	10
Selection Sort	~0.0	~0.01199	~1.21376	~123.607	10
Bubble Sort	~0.0	~0.02498	~2.76620	~295.339	9
Timsort	~0.0	~0.0	~0.0	~0.01061	~0.15076	~2.30560	~53.2125	21

Let's now calculate the weights received from the test with theoretical above:

Quicksort	Merge Sort	Insertion Sort	Selection Sort	Bubble Sort	Timsort
$7 + 16 = \mathbf{23}$	$11 + 16 = \mathbf{27}$	$8 + 10 = \mathbf{18}$	$4 + 10 = \mathbf{14}$	$8 + 9 = \mathbf{17}$	$12 + 21 = \mathbf{33}$

We have got the following top: Timsort (33) – 1 Place; Merge Sort (27) – 2 Place; Quicksort (23) – 3 Place; Insertion Sort (18) – 4 place; Bubble Sort (17) – 5 place; Selection Sort (14) – 6 place.

Conclusion

In this paperwork we considered 6 different sorting methods from the perspective of theory and practice. Overall, the best performance showed Timsort, being precise, efficient and stable, for what it was chosen to be implemented in `sorted()` function in Python and Java. Merge Sort and Quicksort turned out to be quite fast even when the size of the problem significantly grew. Insertion Sort, Selection Sort and Bubble Sort, as expected, didn't show any astonishing results due to their average complexity of $O(n^2)$ that makes them unpractical for sorting huge arrays.

That being said, some of those popular algorithms that are considered very slow can serve as good examples in educational purposes and when it's enough to sort a small array. They are easy to implement, however, it doesn't make them good algorithms, because there are always better options and solutions.

We, as future scientists and engineers, must be as efficient and reasonable as possible with our inventions. I think, if there is an opportunity to solve the task more favorably and faster, we should take it and squeeze 110% of the tools and knowledge that we use to come across success. That might save lives.

Bibliography

- Slides from the ADS course
- <https://ru.wikipedia.org/wiki/Timsort>
- <https://www.geeksforgeeks.org/timsort/>
- Грохаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. - СПб.: Питер, 2017. - 288 с. : ил. - (Серия «Библиотека программиста»). ISBN 978-5-496-02541-6